Lista de Prioridades

Lista de Prioridades

Uma ***lista de prioridade*** é uma tabela na qual a cada um de seus dados está associada uma prioridade (um valor numérico).

Uma estrutura de dados ***heap*** é uma **lista linear** que pode ser visualizada como uma **árvore binária completa**, com o último nível preenchido da esquerda para a direita.

Lista de Prioridades

Operações

→ Selecionar o dado de maior prioridade

→ Inserir um novo elemento

→ Remover o elemento de maior prioridade

→ Modificar a prioridade de um elemento

Lista de Prioridades

Estruturas já conhecidas:

→ Lista não ordenada

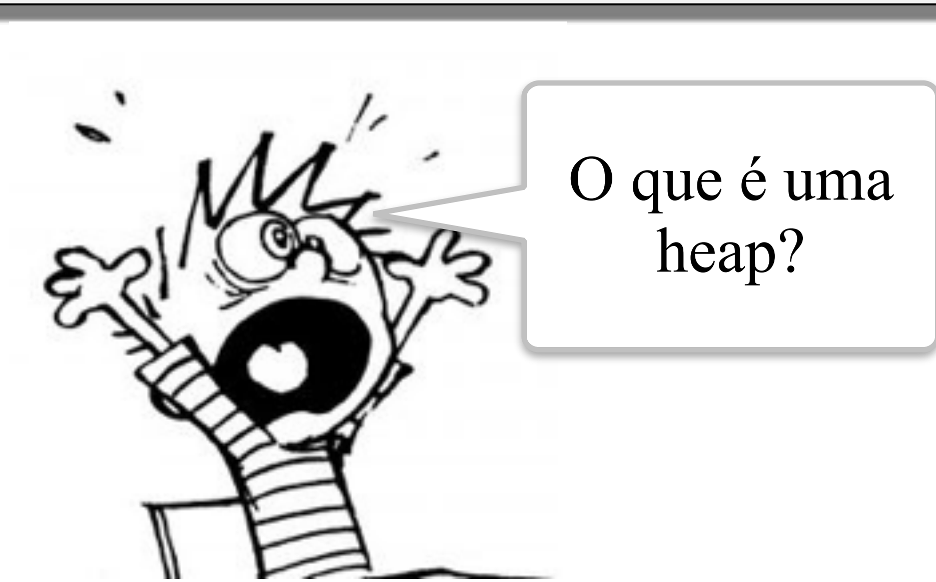
→ Lista ordenada

Dados *n* elementos, qual a complexidade de:

→ Selecionar o de maior prioridade?
→ Inserir um novo elemento?
→ Remover o elemento de maior prioridade?
→ Construir a lista de prioridades?

Lista de Prioridades: Heap

Uma estrutura de dados ***heap*** é uma **lista linear** que pode ser visualizada como uma **árvore binária completa**, preenchida **em nível** da esquerda pra direita.



Lista de Prioridades

Uma *lista de prioridade*, implementada por uma heap, é uma lista de elementos com prioridades s_1, ..., s_n, satisfazendo a seguinte propriedade:

Heap Max

$$s_i \leq s_{\lfloor i/2 \rfloor} \quad \text{para } 1 < i \leq n.$$

Heap Min

$$s_i \geq s_{\lfloor i/2 \rfloor} \quad \text{para } 1 < i \leq n.$$

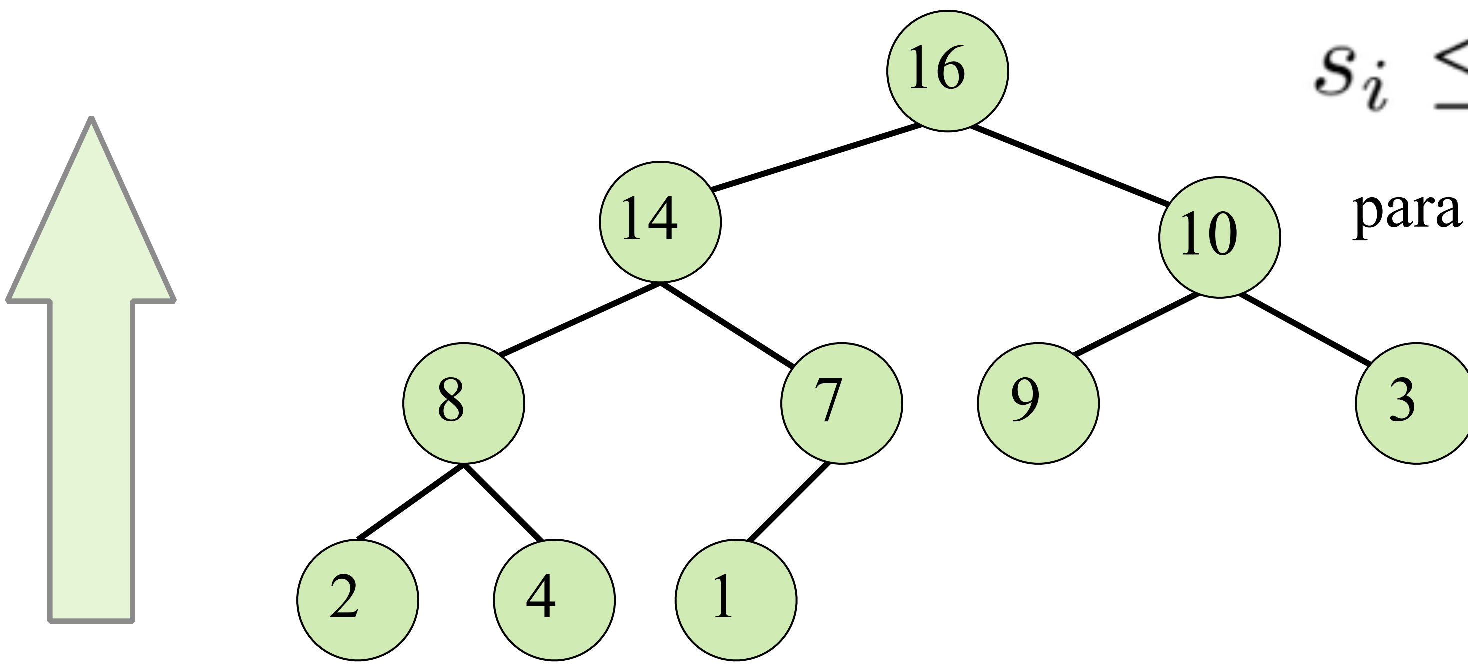
Lista de Prioridades

Heap max

16	**14**	**10**	**8**	**7**	**9**	**3**	**2**	**4**	**1**	**Armazenamento**
1	2	3	4	5	6	7	8	9	10	

$$s_i \leq s_{\lfloor i/2 \rfloor}$$

para $1 < i \leq n$.



Visualização

Lista de Prioridades

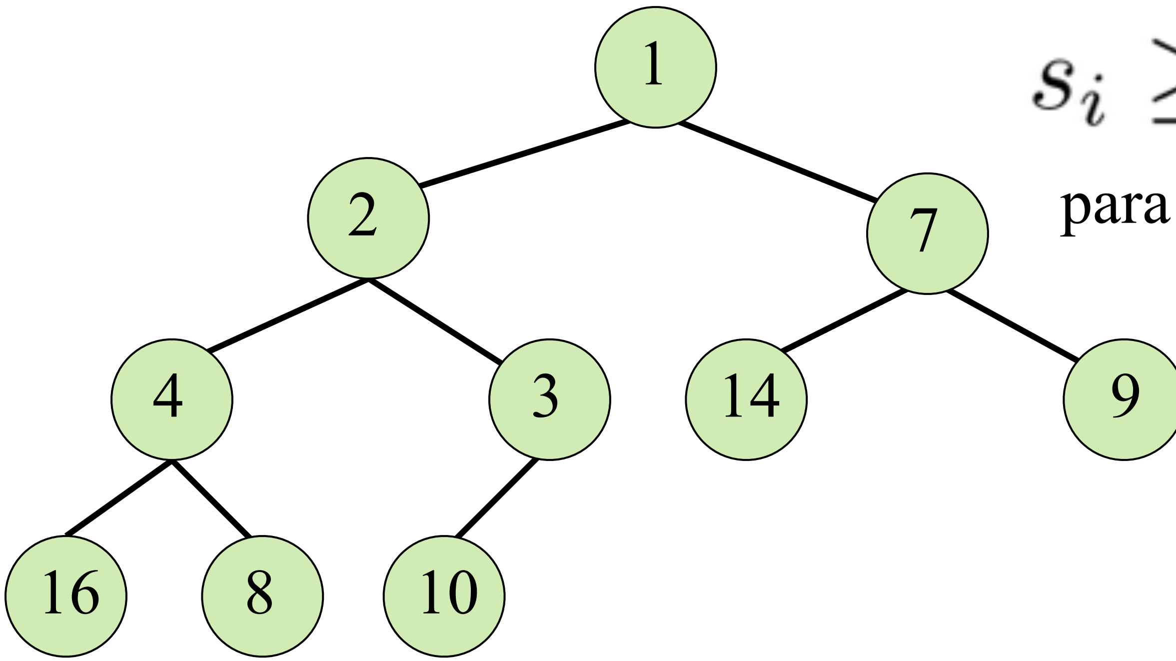
Heap min

1	**2**	**7**	**4**	**3**	**14**	**9**	**16**	**8**	**10**
1	2	3	4	5	6	7	8	9	10

Armazenamento

$$s_i \geq s_{\lfloor i/2 \rfloor}$$

para $1 < i \leq n$.



Visualização

Cálculo dos Índices

H é o vetor que armazena a heap.
H tem comprimento *MAX* e a heap tem comprimento *n*,
$n \leq MAX$.

A **raiz** da árvore é **H[1]**. Dado *i* o índice de um nó, os índices do seu pai e filhos são dados por:

$$\mathbf{Pai}(i) = i \,/\, 2$$

$$\mathbf{Filho\_esq}(i) = 2i$$

$$\mathbf{Filho\_dir}(i) = 2i + 1$$

Exercício

Verifique se as sequências abaixo correspondem a um heap max(min):

a) 20, 25, 32, 29, 27, 35, 40, 45

b) 20, 32, 25, 29, 27, 35, 40, 45

c) 50, 45, 48, 29, 15, 35, 40, 27

Heap: Lema 1

PROPRIEDADE IMPORTANTE

Lema 1

Se T é uma heap qualquer com n elementos, então T possui $\lceil n/2 \rceil$ folhas.

Prova:

Se ***n* é par**, o $\lfloor (n/2) \rfloor$-*ésimo* elemento possui apenas um filho (esquerdo), a saber, o *n-ésimo* elemento.

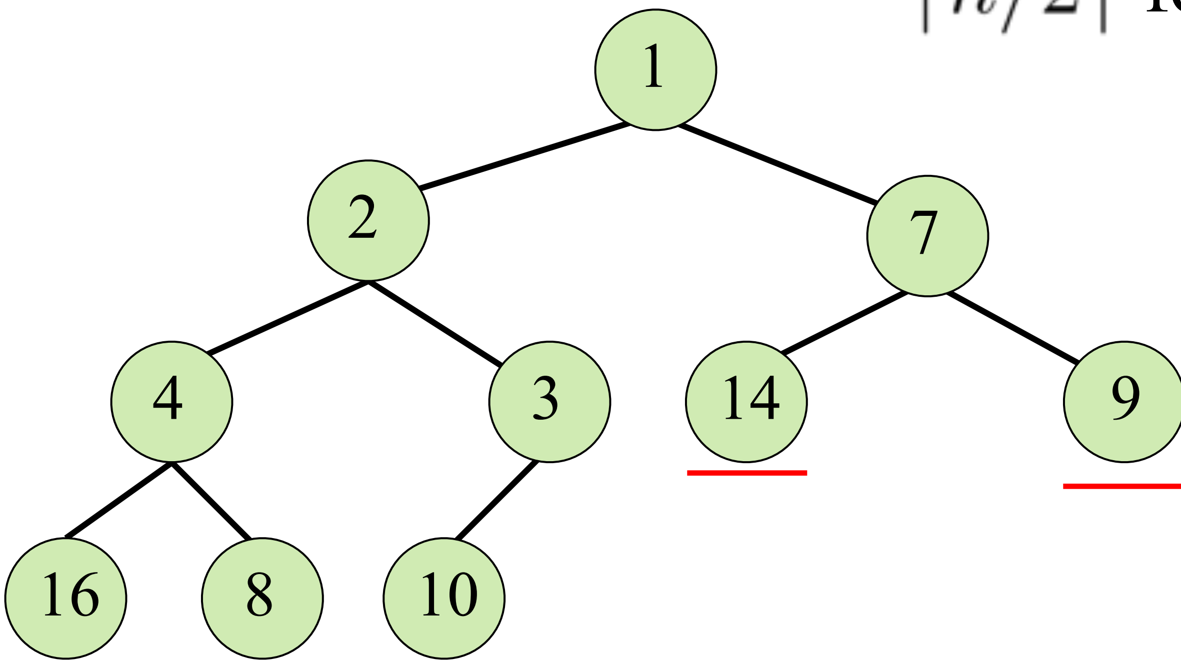
Se ***n* é impar**, o $\lfloor (n/2) \rfloor$-*ésimo* elemento possui um filho esquerdo (índice n-1) e um direito (índice n).

Em ambos os casos, os últimos $\lceil n/2 \rceil$ são folhas.

Da matemática discreta, você deve lembrar que: $\lfloor n/2 \rfloor + \lceil n/2 \rceil = n$

Heap: Lema 1 (Exemplo)

$\lceil n/2 \rceil$ folhas



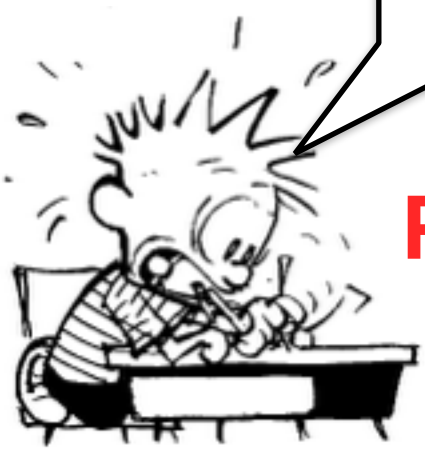
1	**2**	**7**	**4**	**3**	**14**	**9**	**16**	**8**	**10**
1	2	3	4	5	6	7	8	9	10

Heap: Lema 2

PROPRIEDADE SUUUUUUUUPER IMPORTANTE!!!!!!!

Lema 2

Se T é uma heap qualquer com n elementos, então T possui, no máximo, $\lceil n/2^h \rceil$ nós com altura h.



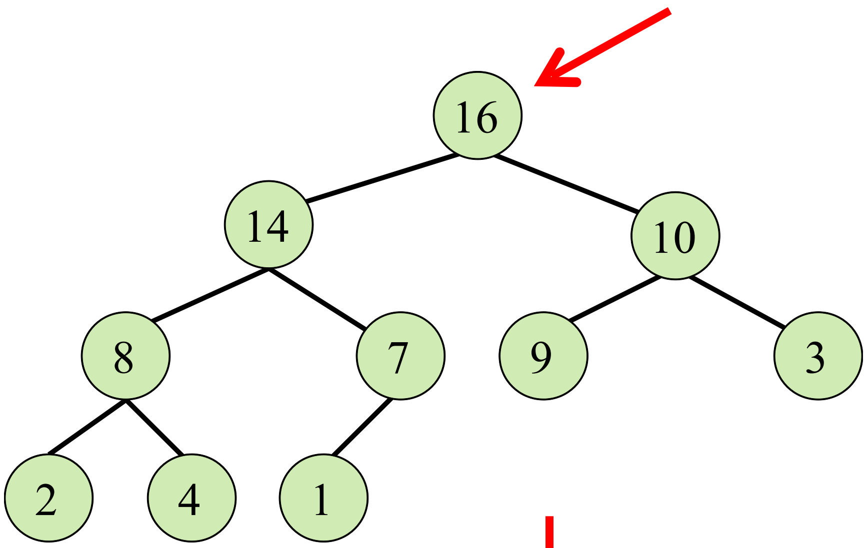
Prova: EXERCÍCIO

Dica: utilize indução matemática e adote o Lema 1.

Remoção

Remove o elemento de maior (menor) prioridade: H[1]

A lista tem que ser ajustada e manter as propriedades



16	14	10	8	7	9	3	2	4	1

Remoção

Como fazer isto?

Remoção

Remove o elemento de maior (menor) prioridade: H[1]

O último elemento substitui o primeiro e será usado o procedimento ***descer*** para ajustar este elemento na heap novamente.

Procedimento
descer?

Descer (heap max)

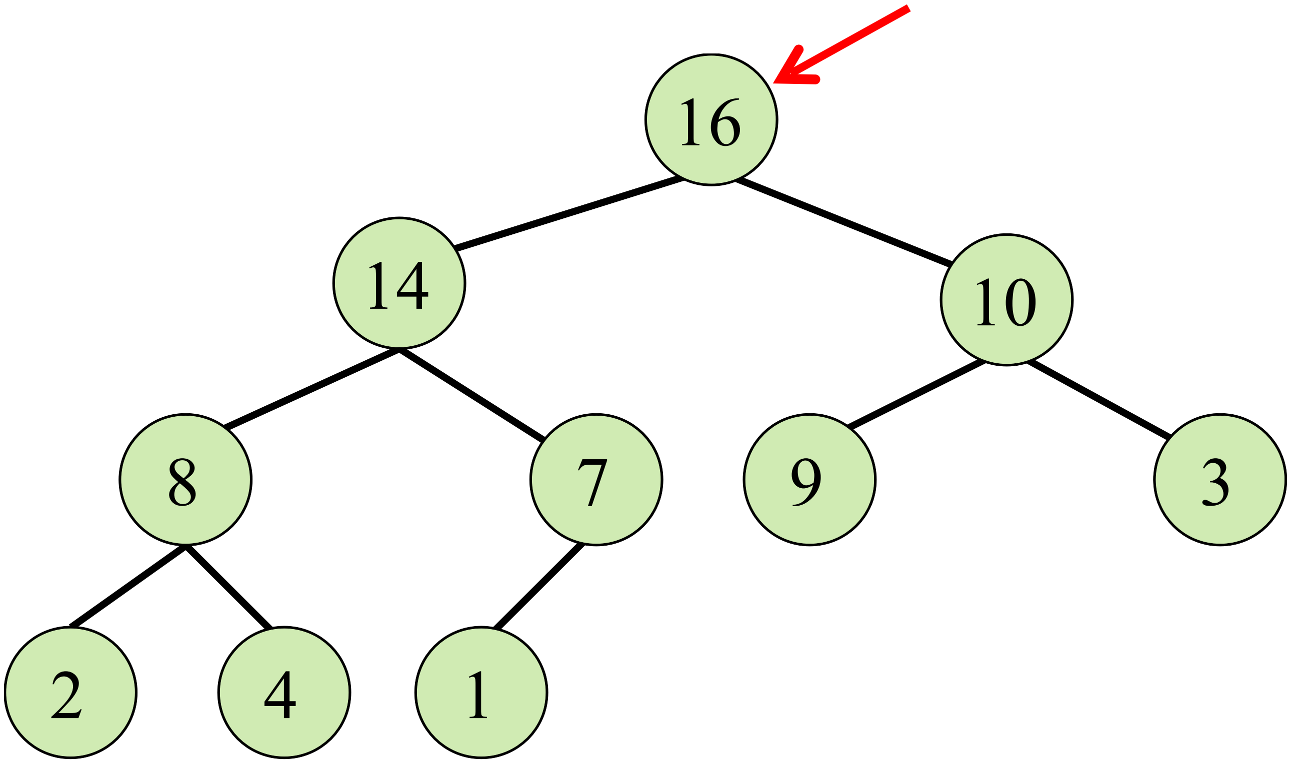
Filho esquerdo de i

descer (i,n)
    j ← esq(i)
    se j ≤ n então
        se j<n então
            se H[j+1] > H[j] então j ← j + 1
        se H[i] < H[j] então
            troca(i,j)
            descer(j,n)

Esta linha verifica qual o filho de *i* que possui maior prioridade

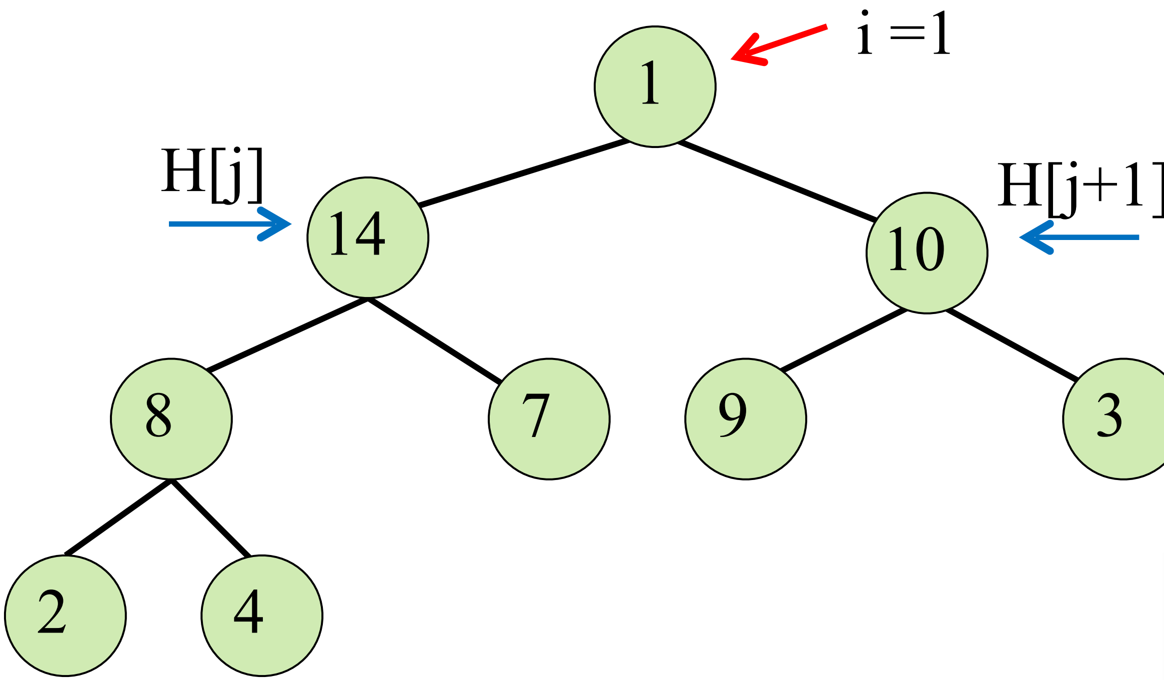
Chamada recursiva

Remoção



16	**14**	**10**	**8**	**7**	**9**	**3**	**2**	**4**	**1**

Descer (heap max)



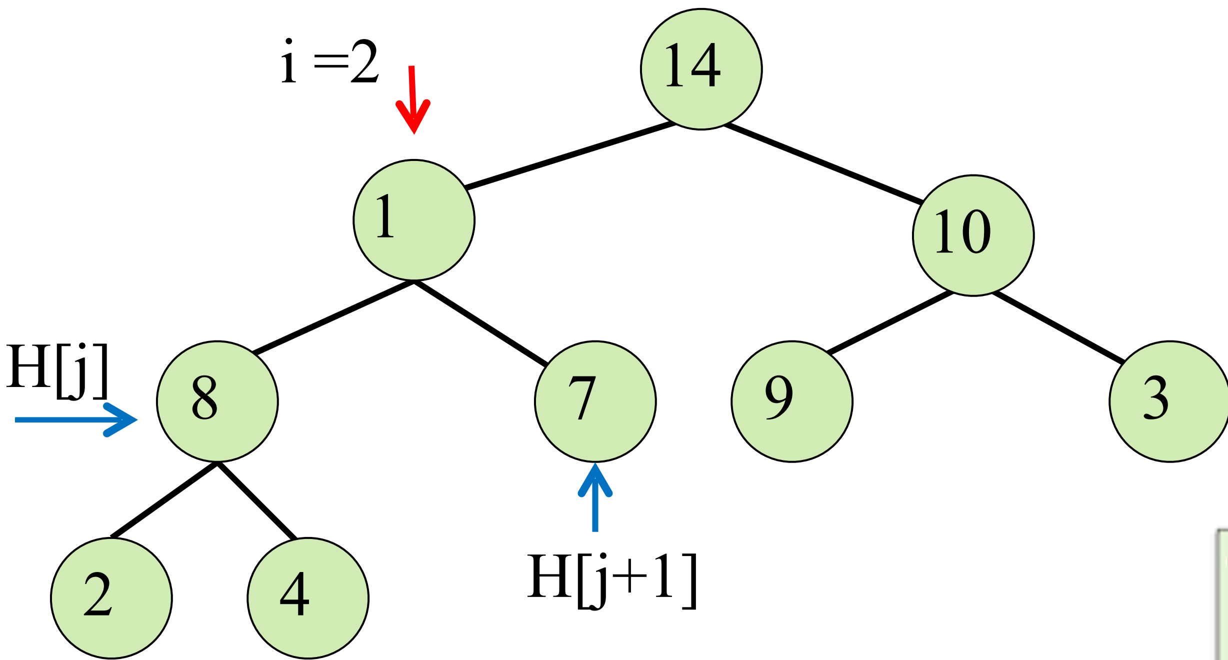

```
descer (i,n)
   j ← esq(i)
   se j ≤ n então
       se H[j+1] > H[j] então j ← j + 1
       se H[i] < H[j] então
           troca(i,j)
           descer(j,n)
```

$j = 2$

1	14	10	8	7	9	3	2	4	

$n = 9$

Descer (heap max)



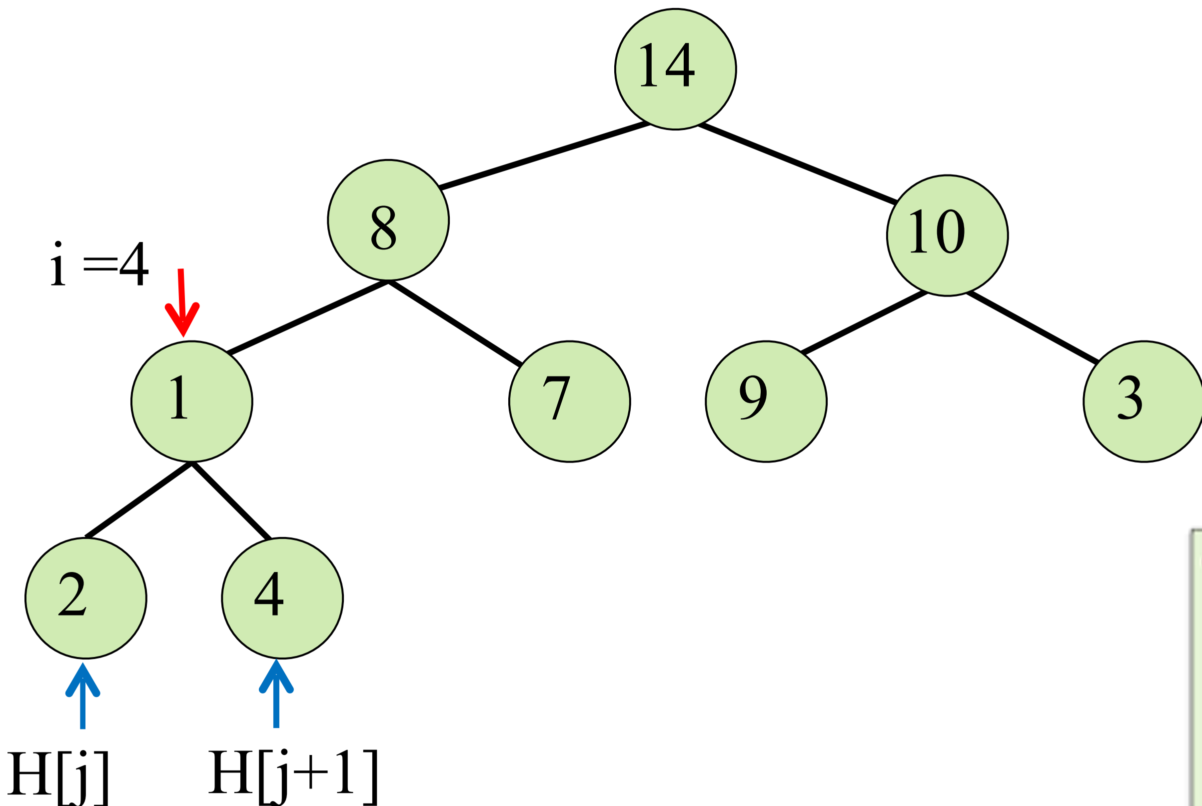

```
descer (i,n)
   j ← esq(i)
   se j ≤ n então
      se H[j+1] > H[j] então j ← j + 1
      se H[i] < H[j] então
         troca(i,j)
         descer(j,n)
```

$j = 4$

14	**1**	**10**	**8**	**7**	**9**	**3**	**2**	**4**	

$n = 9$

Descer (heap max)



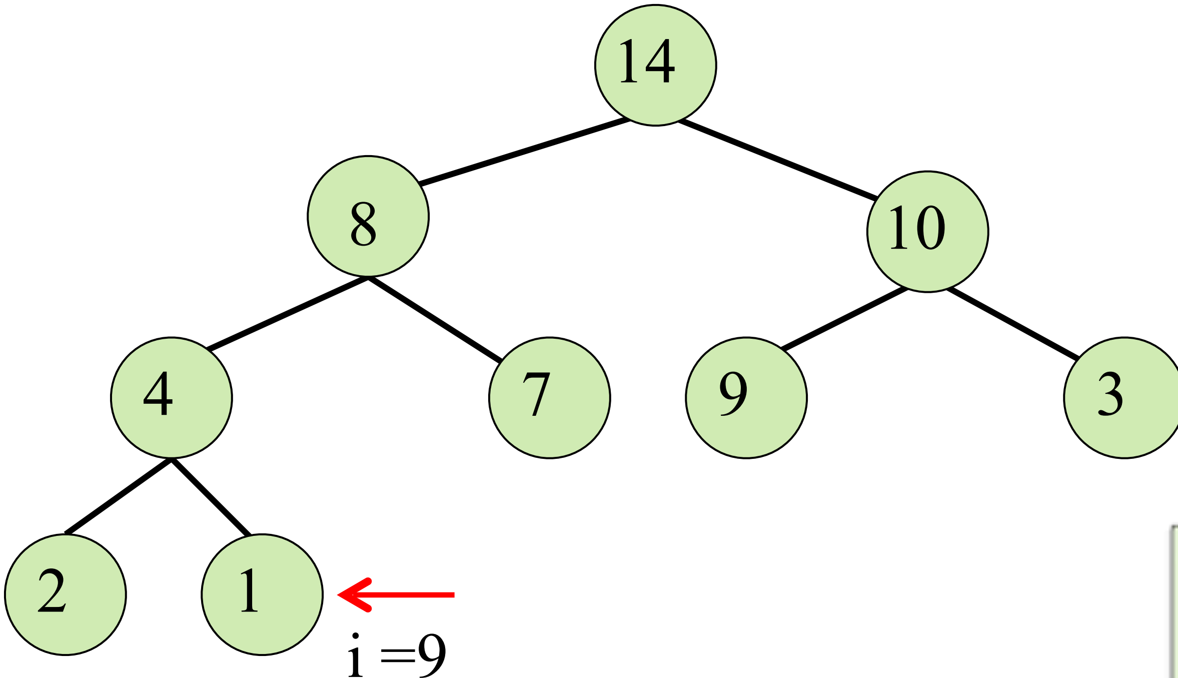

```
descer (i,n)
   j ← esq(i)
   se j ≤ n então
      se H[j+1] > H[j] então j ← j + 1
      se H[i] < H[j] então
         troca(i,j)
         descer(j,n)
```

$j = 9$

14	**8**	**10**	**1**	**7**	**9**	**3**	**2**	**4**	

$n = 9$

Descer (heap max)



$\boxed{j = 18 \quad \text{FIM}}$

```
descer (i,n)
   j ← esq(i)
   se j ≤ n então
      se H[j+1] > H[j] então j ← j + 1
      se H[i] < H[j] então
         troca(i,j)
         descer(j,n)
```

14	**8**	**10**	**4**	**7**	**9**	**3**	**2**	**1**	

n =9

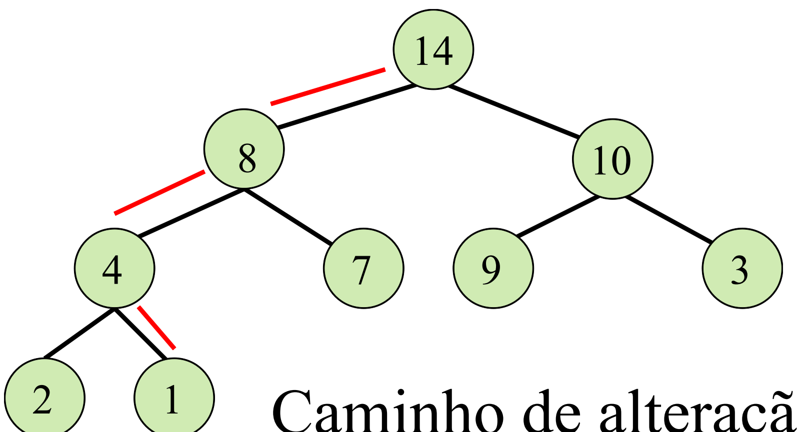
Descer

Qual a complexidade de *descer*?

Descer

14	8	10	4	7	9	3	2	1	

Posições inalteradas



Caminho de alteração

Altura de uma árvore binária completa

Descer

Qual a complexidade de *descer*?

Qual a altura da árvore completa mesmo?

Lema 2
Considere T uma árvore binária completa com $n > 0$ nós.
Então T possui altura h mínima.
Além disso, $h = 1 + \lfloor \log n \rfloor$.

Descer

Qual a complexidade de *descer*?

$$O(\log n)$$

Remoção

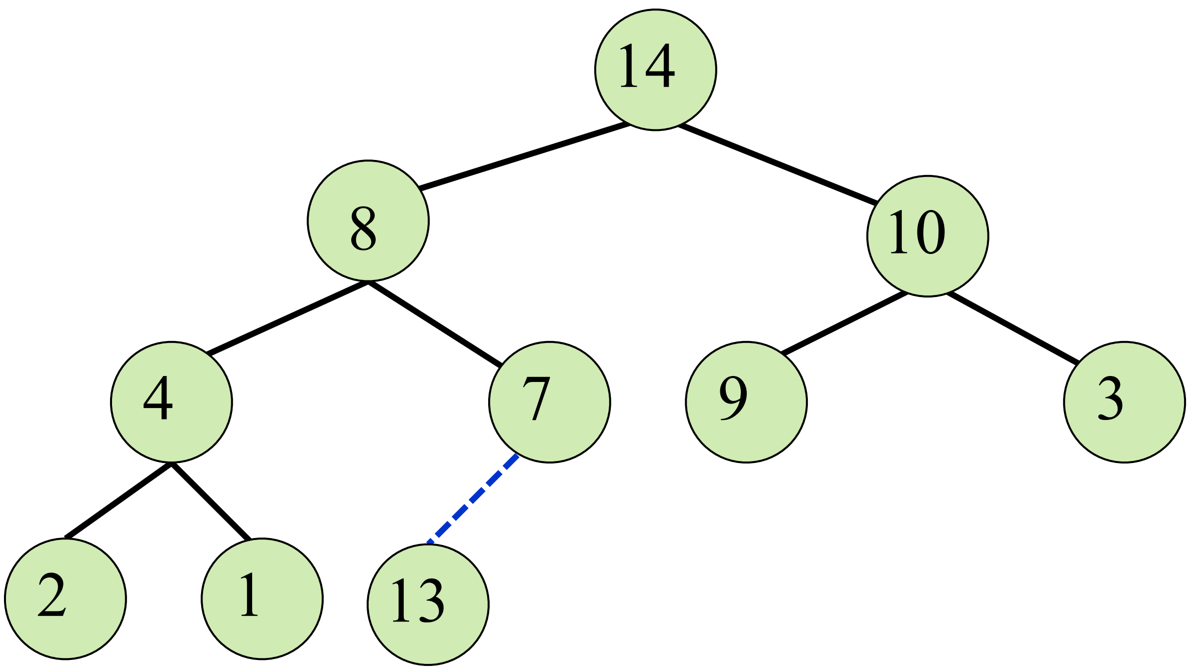
Remoção
    se ($n > 0$) então
        elemento $\leftarrow$ H[1]
        H[*1*] $\leftarrow$ H[*n*]
        $n \leftarrow n - 1$
        descer(1,*n*)
   senão "Heap Vazia"

Qual a complexidade da *remoção* do elemento de maior prioridade?

$$\Theta(\log n)$$

Inserção

O novo dado é inserido na posição ***n*+1** e “subido” se necessário



Inserir o novo elemento na posição $n+1$

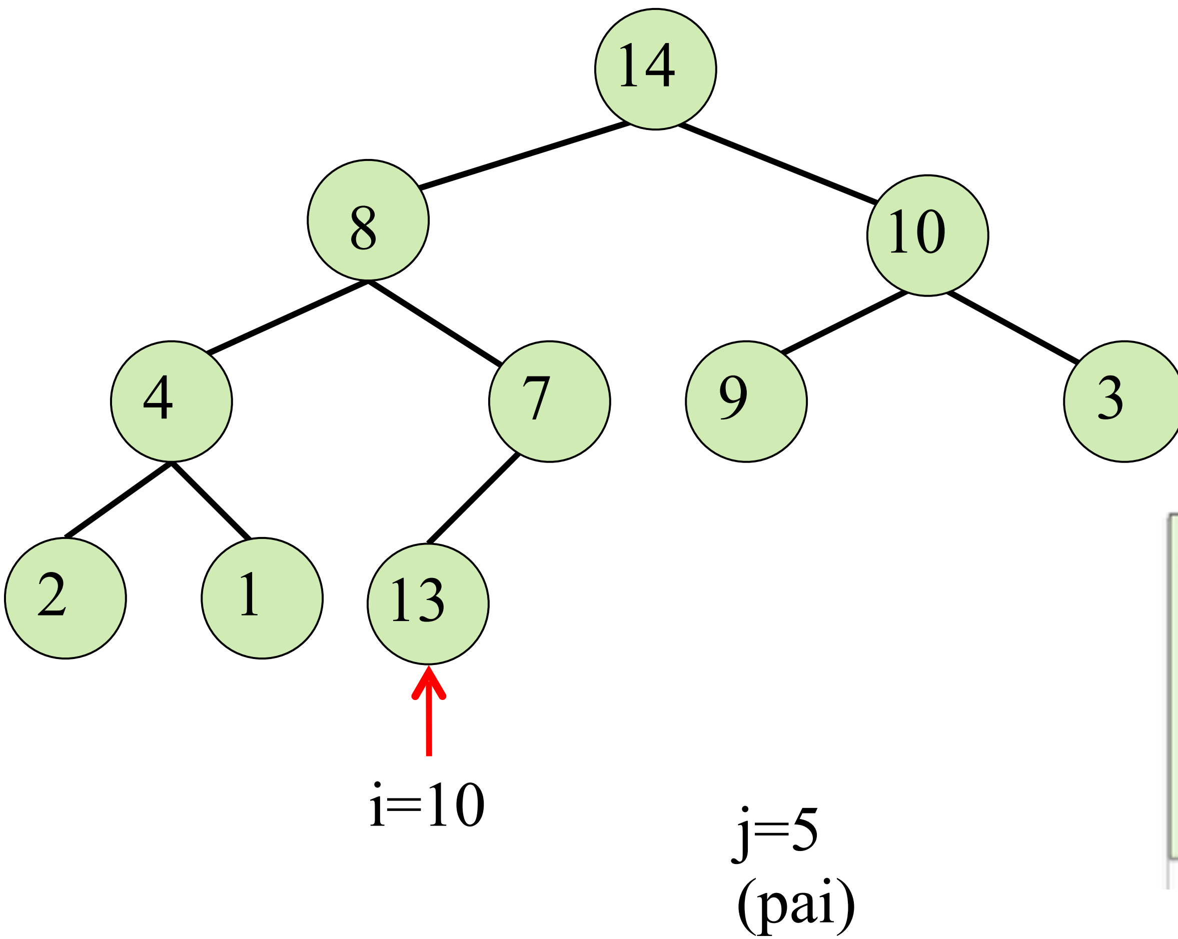
14	8	10	4	7	9	3	2	1	13

Subir (heap max)

```
subir (i)
    j ← pai(i)
    se j ≥ 1 então
        se H[i] > H[j] então
            troca(i,j)
            subir(j)
```

Verifica se a prioridade do filho *i* é maior que a do pai *j*

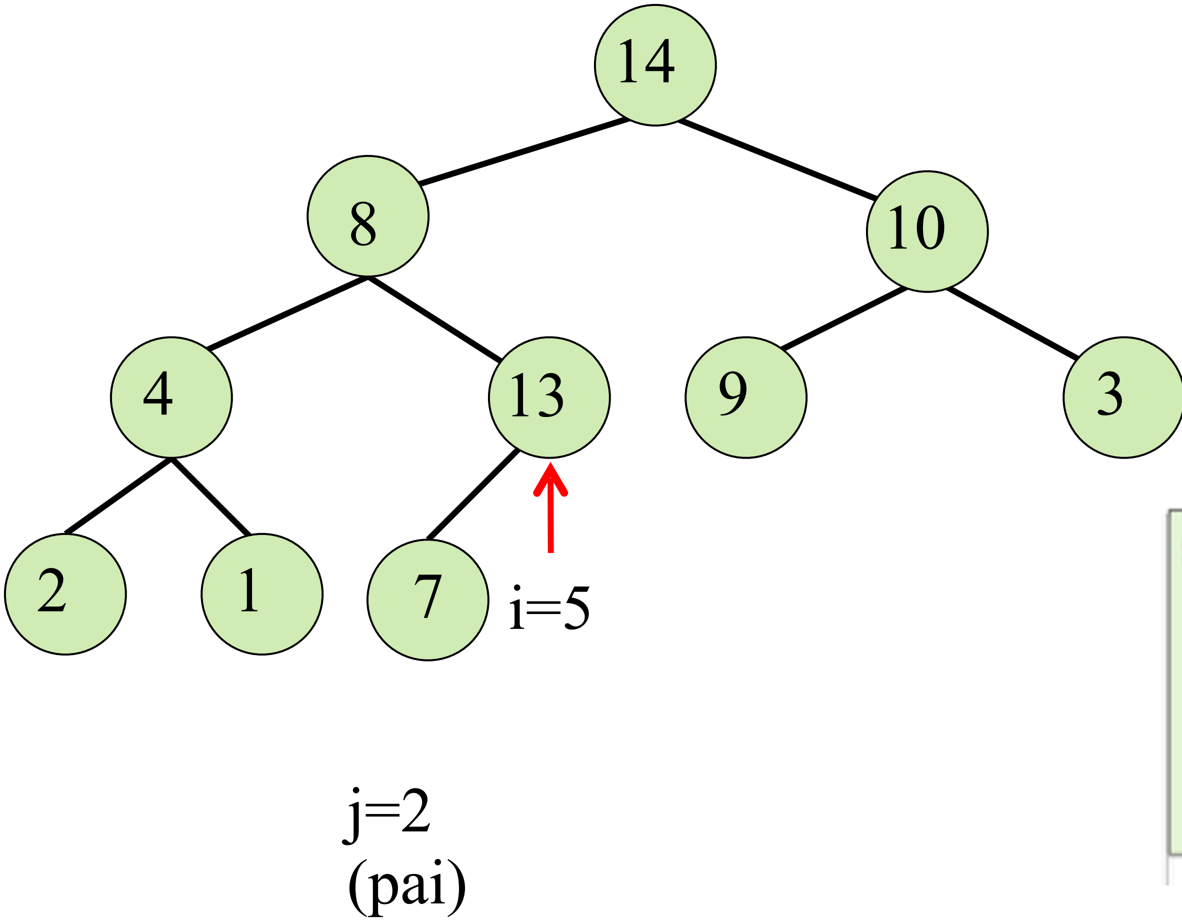
Subir (heap max)




```
subir (i)
    j ← pai(i)
    se j ≥ 1 então
        se H[i] > H[j] então
            troca(i,j)
            subir(j)
```

14	8	10	4	7	9	3	2	1	13

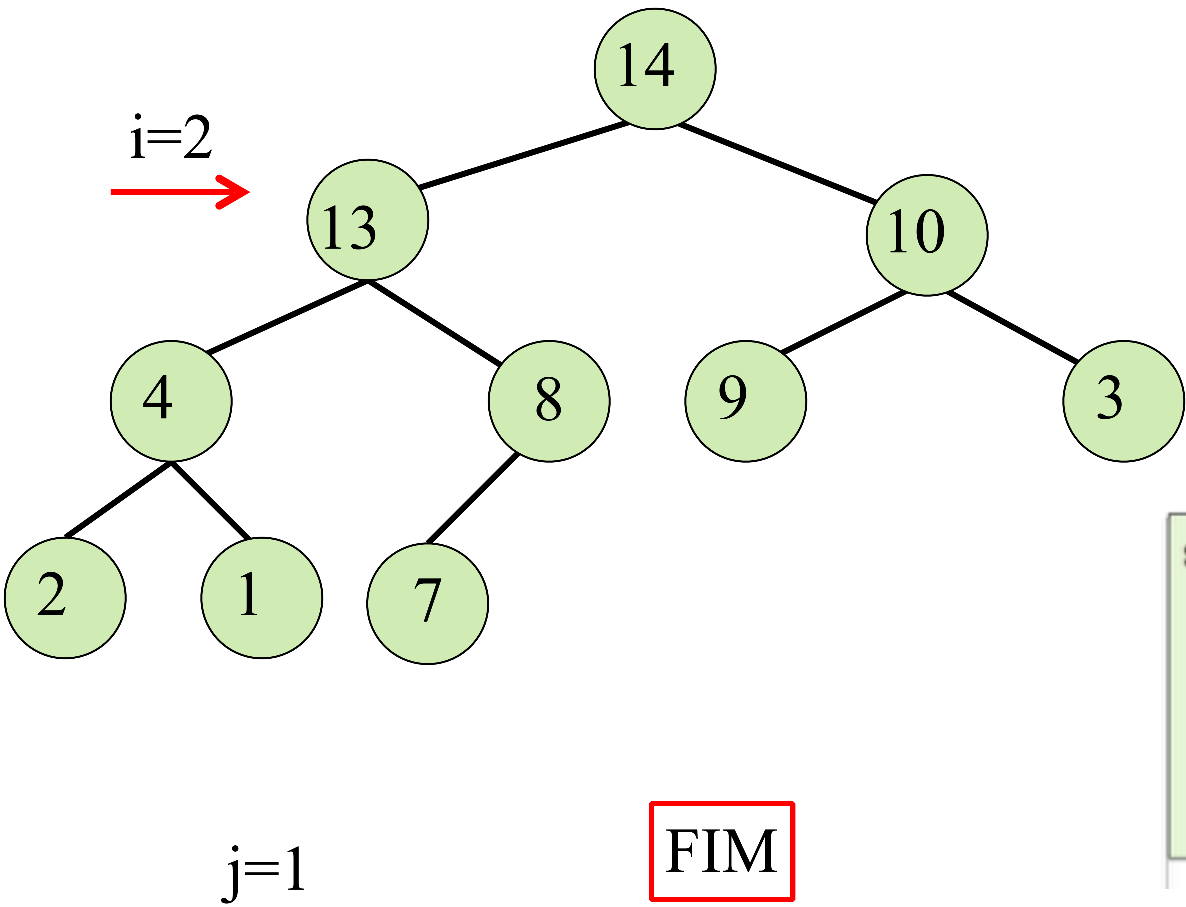
Subir (heap max)




```
subir (i)
   j ← pai(i)
   se j ≥ 1 então
      se H[i] > H[j] então
         troca(i,j)
         subir(j)
```

14	8	10	4	13	9	3	2	1	7

Subir (heap max)



FIM

```
subir (i)
    j ← pai(i)
    se j ≥ 1 então
        se H[i] > H[j] então
            troca(i,j)
            subir(j)
```

14	13	10	4	8	9	3	2	1	7

Subir

Qual a complexidade de *subir*?

Subir

Qual a complexidade de *subir*?

$$O(\log n)$$

Inserção

Inserção
    se ($n < MAX$) então
        $n \leftarrow n + 1$
        H[n] $\leftarrow$ *novo*
        subir(n)
    senão “Não há espaço”

Complexidade da *inserção*: $O(\log n)$

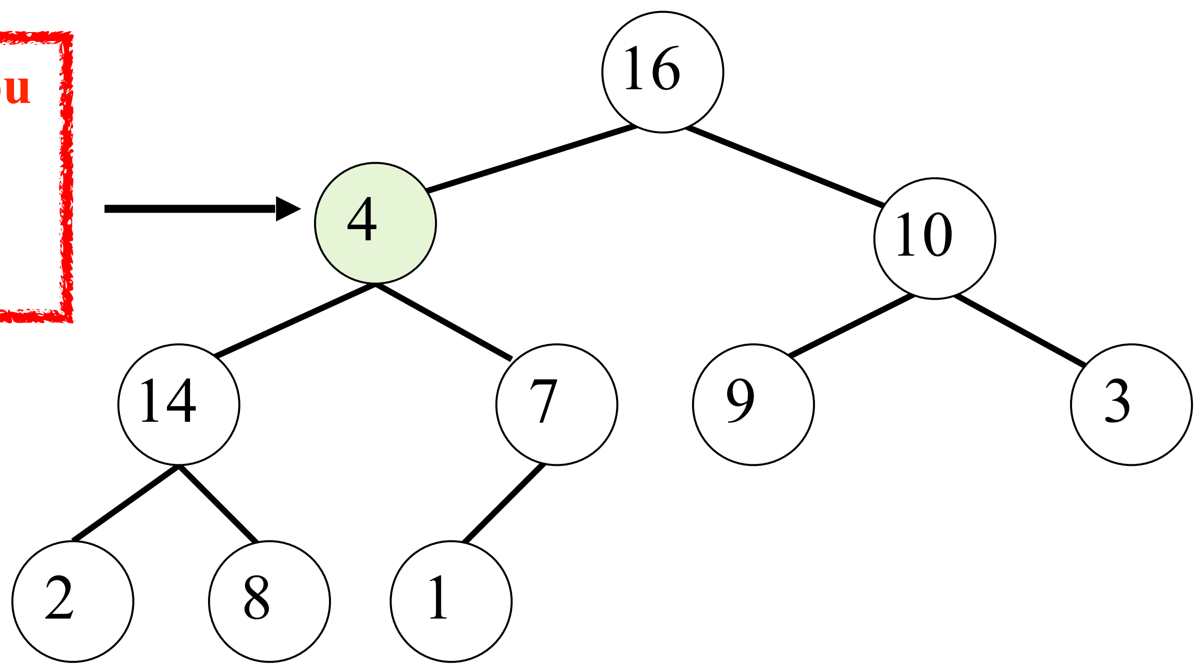
Exercício

Escrever os procedimentos subir e descer não recursivos. Qual a complexidade destes procedimentos?

Alteração de Prioridade

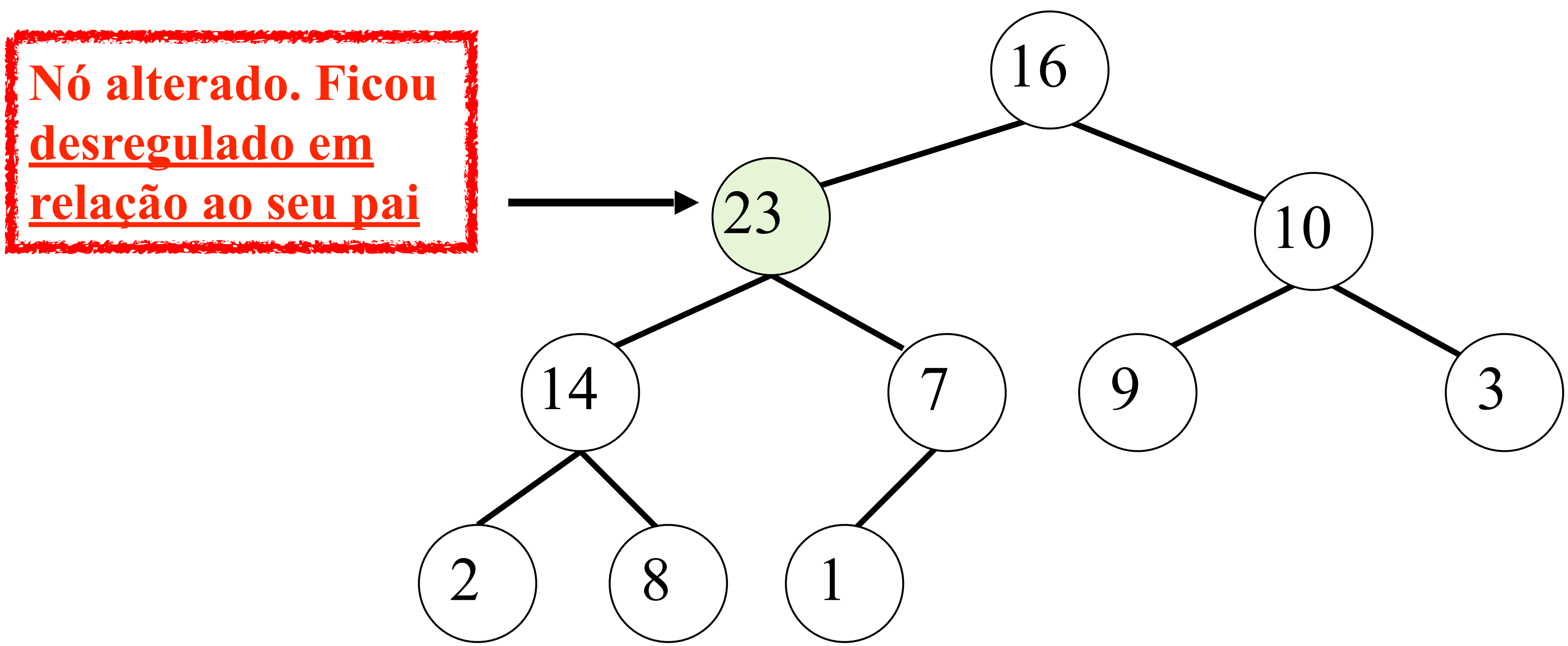
Suponha que uma prioridade H[*i*] tenha sido modificada e tenha ficado **menor** que a prioridade dos seus filhos. Isto contraria a propriedade da **heap max**. Sendo assim, H[*i*] deve "**descer**" por um caminho na árvore até que a propriedade seja satisfeita.

Nó alterado. Ficou <u>desregulado em relação aos seus filhos.</u>



Alteração de Prioridade

Suponha que uma prioridade H[*i*] tenha sido modificada e tenha ficado **maior** que a prioridade do pai. Isto contraria a propriedade da **heap max**. Sendo assim, H[*i*] deve "**subir**" por um caminho na árvore até que a propriedade seja satisfeita.



Alteração de Prioridade

Exercício

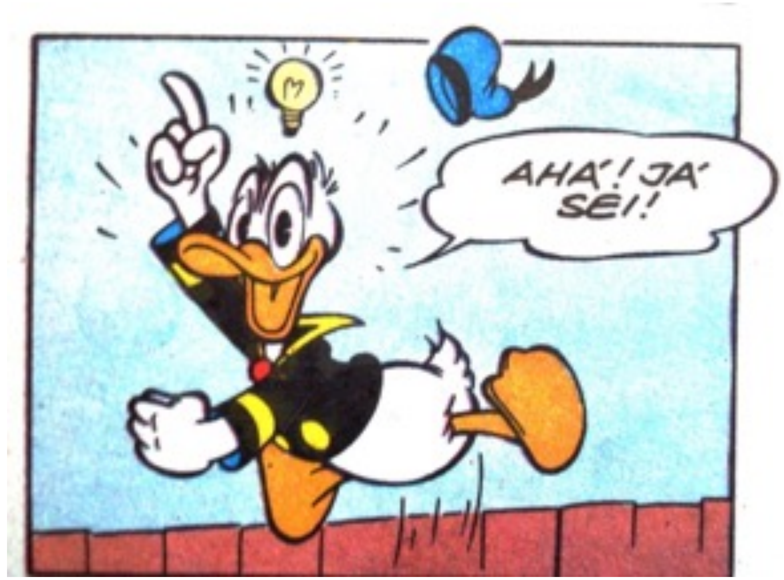
Fazer o algoritmo de alteração de prioridade.

Qual a complexidade do seu algoritmo?

Construção (heap max)

Dado um vetor com *n* elementos dispostos de modo qualquer, como construir uma *heap max*?

Construção (heap max)



Ideia 1: Olhar os elementos um a um e considerar uma nova inserção.

arranjar1 (n)
    para i ← 2 até n faça
        subir(i)

→ **Complexidade: $O(n\log n)$**

Construção (heap max)

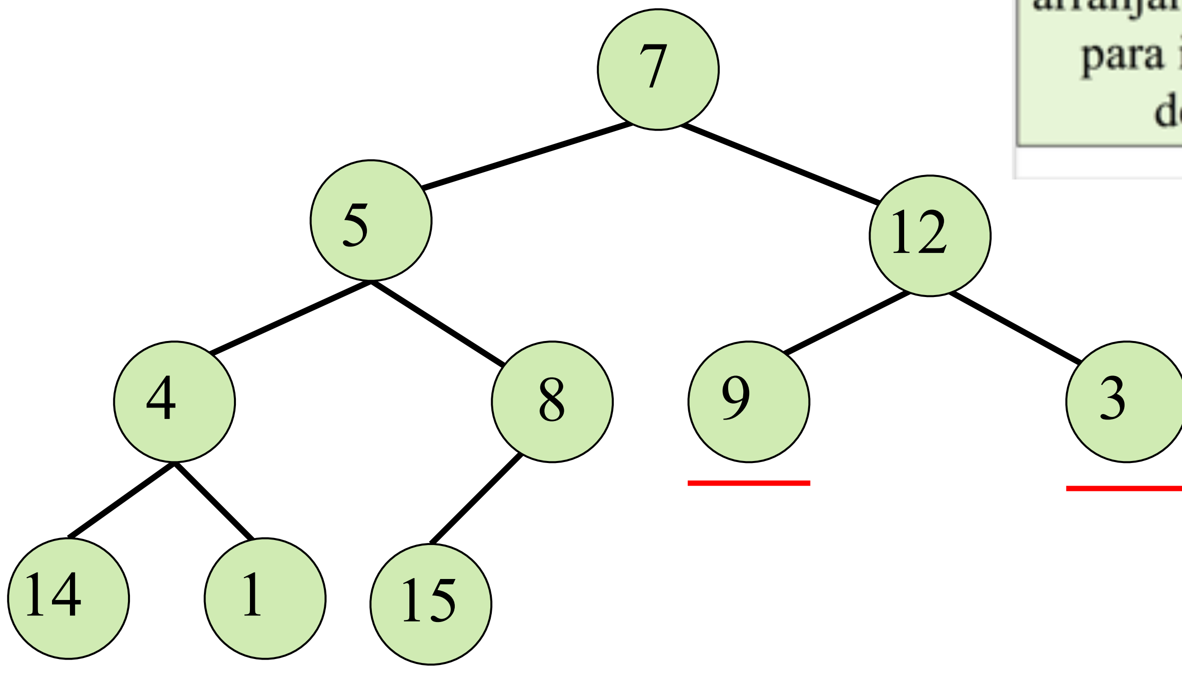
Ideia 2: Não é necessário preocupar-se com elementos ***folhas***, dado que a propriedade da heap estará trivialmente satisfeita para eles. Assim, deve-se, arranjar os $\lfloor n/2 \rfloor$ nós não folha

```
arranjar2 (n)
   para i ← ⌊n/2⌋ até 1 faça
        descer(i,n)
```

→ **Complexidade: ?**

Construção (heap max)

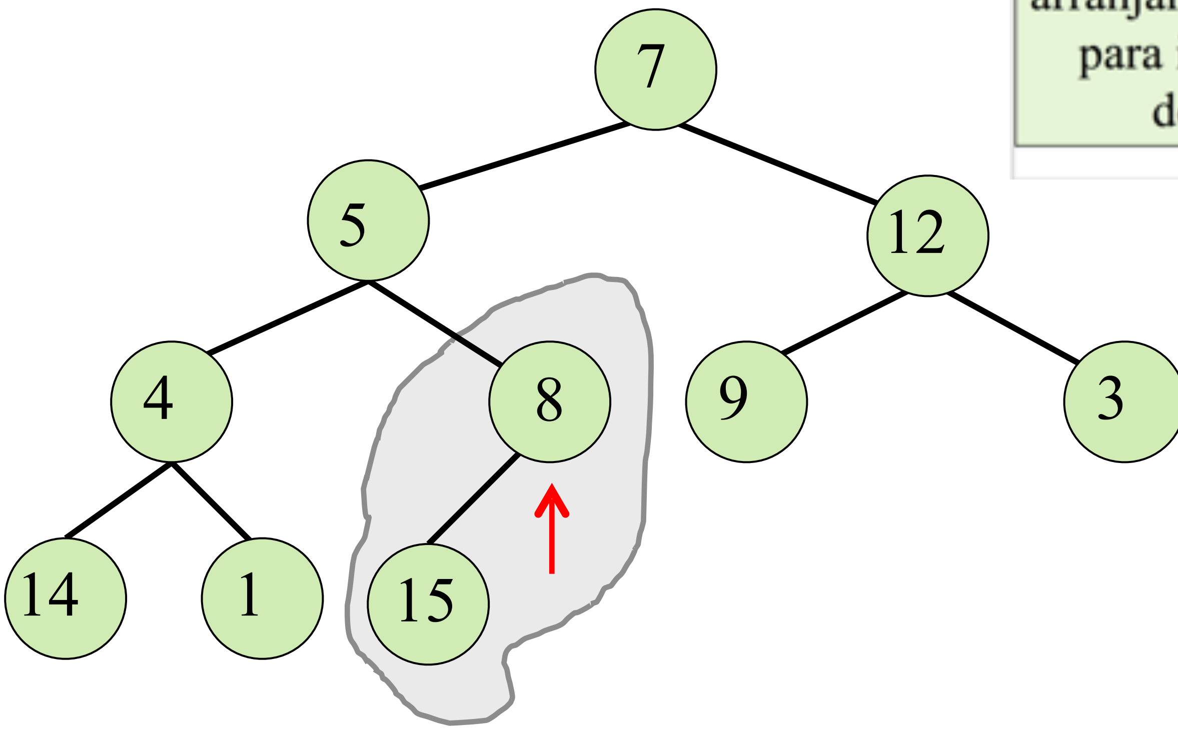
```
arranjar (n)
    para i ← ⌊n/2⌋ até 1 faça
        descer(i,n)
```



7	5	12	4	8	9	3	14	1	15

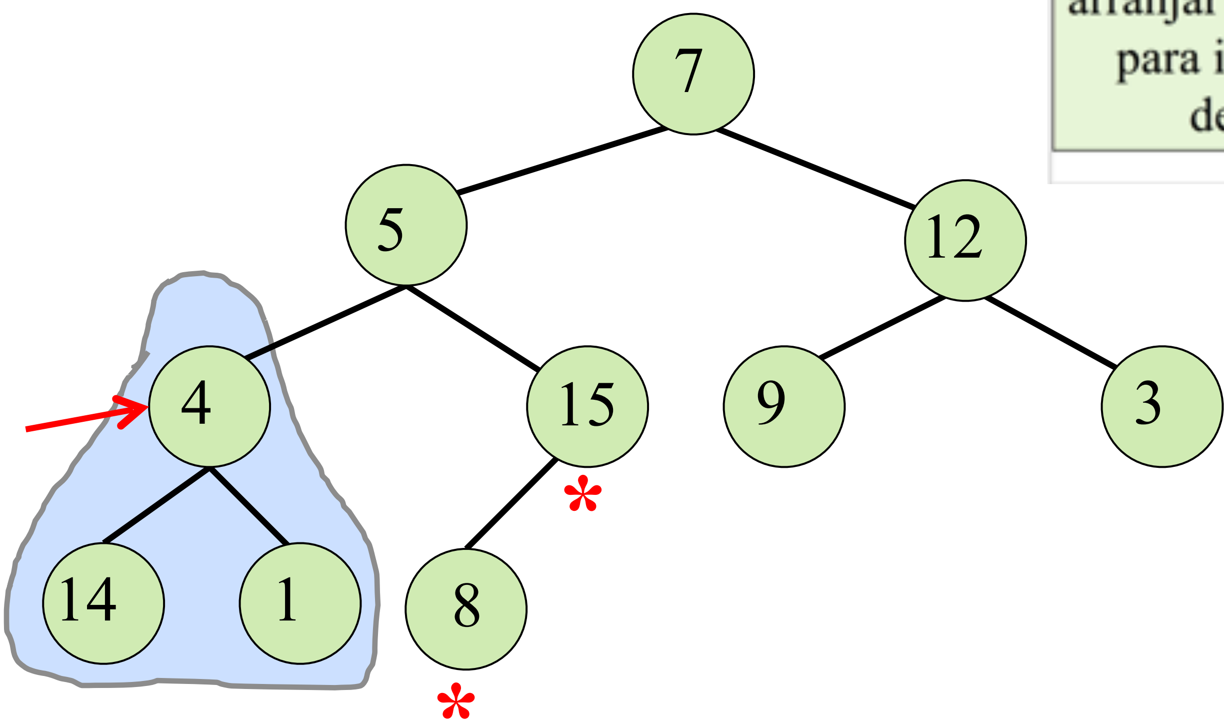
Construção (heap max)

arranjar (n)
  para i ← $\lfloor n/2 \rfloor$ até 1 faça
    descer(i,n)



7	5	12	4	8	9	3	14	1	15

Construção (heap max)

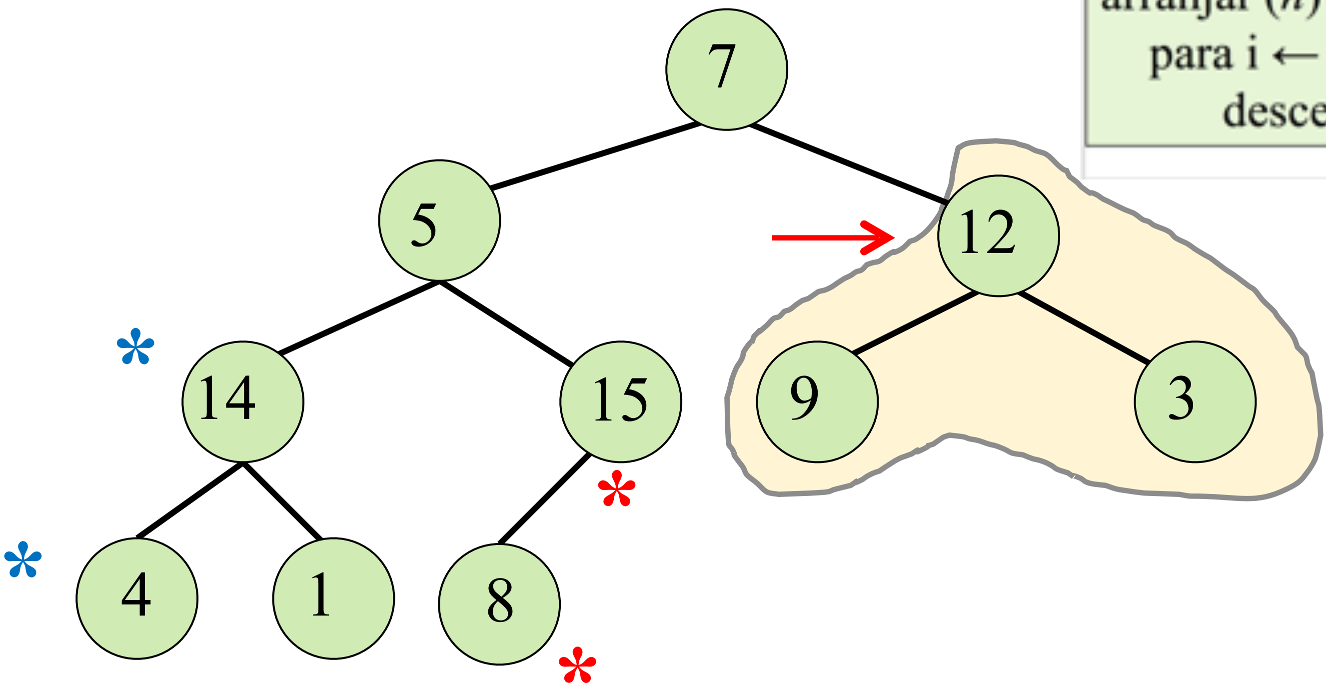


arranjar (n)
  para i ← $\lfloor n/2 \rfloor$ até 1 faça
    descer(i,n)

7	5	12	4	15	9	3	14	1	8

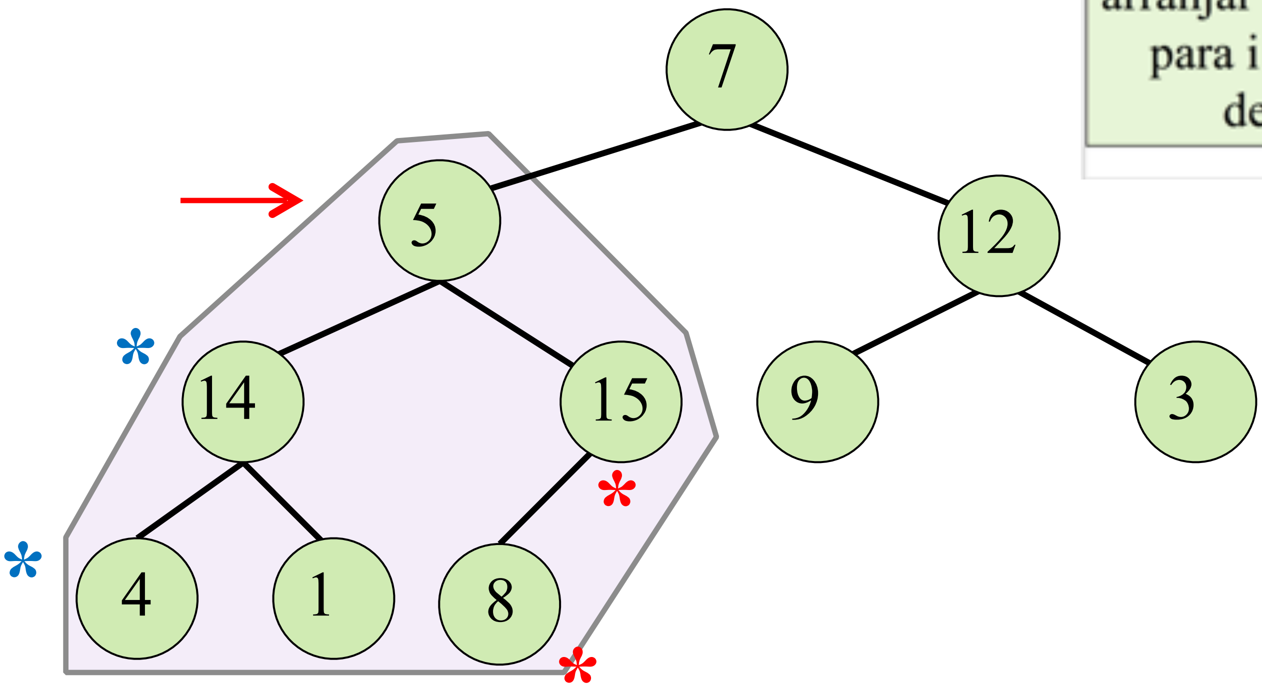
Construção (heap max)

arranjar (n)
  para i ← $\lfloor n/2 \rfloor$ até 1 faça
    descer(i,n)



7	5	12	14	15	9	3	4	1	8

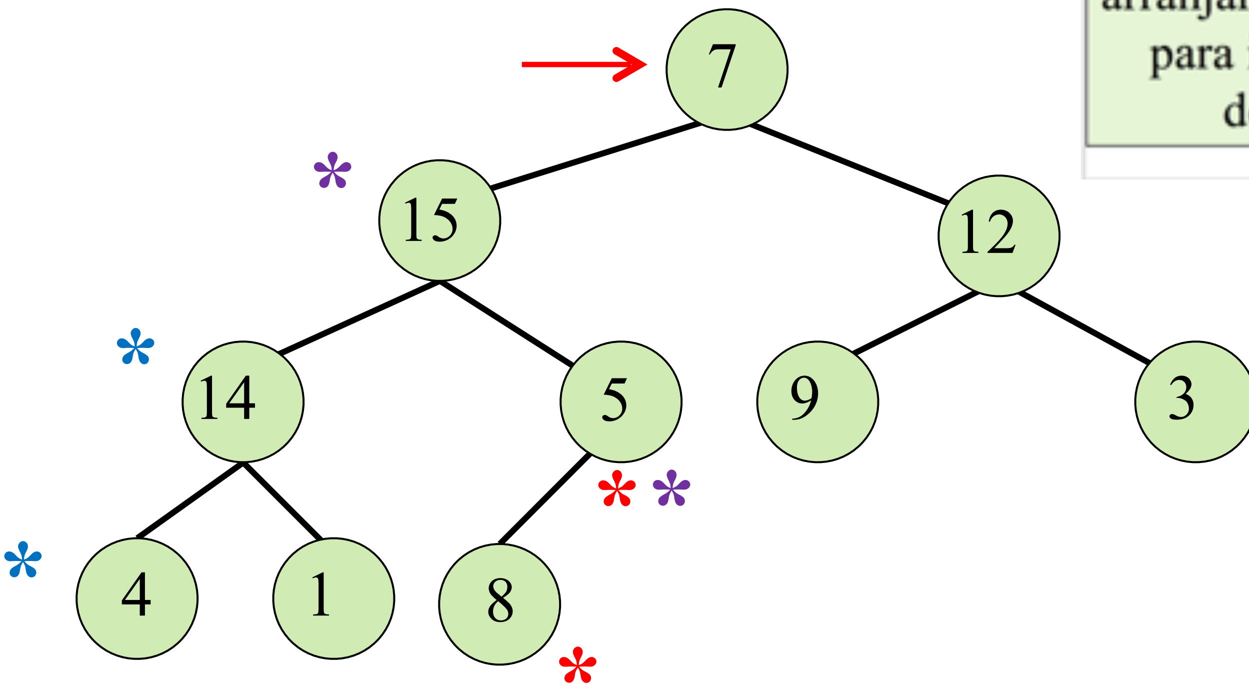
Construção (heap max)



arranjar (n)
    para i ← $\lfloor n/2 \rfloor$ até 1 faça
        descer(i,n)

7	5	12	14	15	9	3	4	1	8

Construção (heap max)

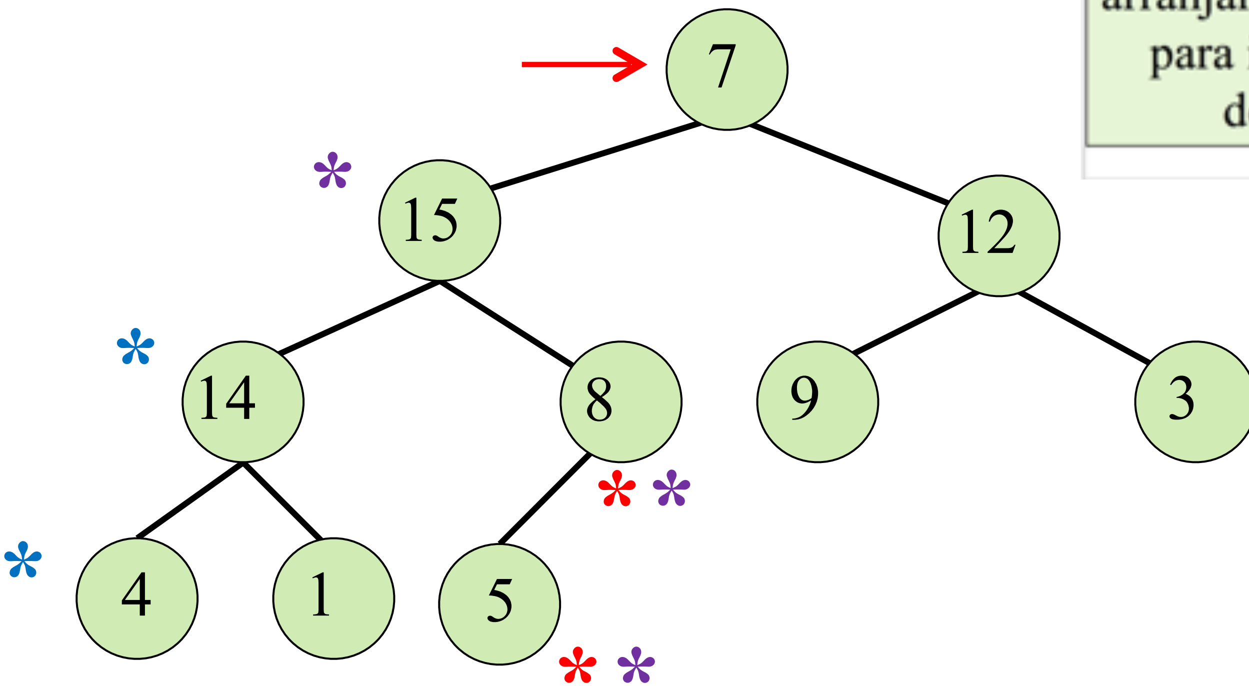



```
arranjar (n)
    para i ← ⌊n/2⌋ até 1 faça
        descer(i,n)
```

7	15	12	14	5	9	3	4	1	8

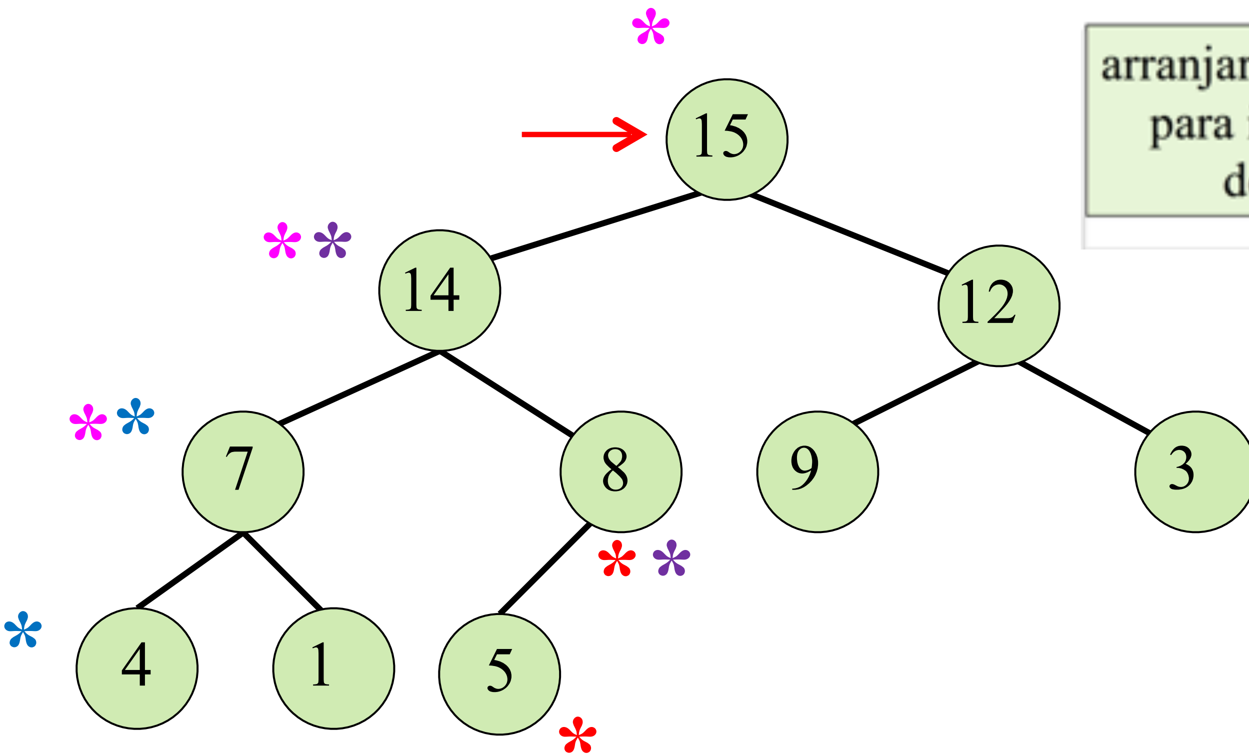
Construção (heap max)

arranjar (n)
    para i $\leftarrow \lfloor n/2 \rfloor$ até 1 faça
        descer(i,n)



7	15	12	14	8	9	3	4	1	5

Construção (heap max)



arranjar (n)
    para i $\leftarrow \lfloor n/2 \rfloor$ até 1 faça
        descer(i,n)

15	14	12	7	8	9	3	4	1	5

Construção (heap max)

<u>**Teorema 1**</u>. O algoritmo ***arranjar2*** constrói uma lista de prioridades em tempo linear.

Prova. O procedimento ***descer*** consome *O(h)* passos para fazer descer um **elemento** de altura *h*. Tal procedimento é chamado apenas para nós com **altura maior que 1 (nós não folha).**

Pelo ***Lema 2***, observa-se que no máximo $\lceil n/2^h \rceil$ elementos possuem altura *h*. O tempo total despendido é:

$$\sum_{h=2}^{logn} h\frac{n}{2^h} = \frac{3n}{2} - logn - 2$$

$$= O(n).$$

Construção: Exercício

Seja a lista abaixo

40, 25, 72, 13, 14, 95, 48, 100, 12, 93.

Determinar o heap max obtido pelo algoritmo de construção.

Heapsort

1. Organizar os elementos como uma ***heap max*** (procedimento ***arranjar2***).

2. Uma vez que o elemento de maior prioridade se encontra na primeira posição, ***H[1]***, basta remove-lo (colocando na última posição).

3. Repetir o processo para os ***n-1*** elementos do heap. Isto deve ser feito até o heap ter tamanho 2.

Heapsort

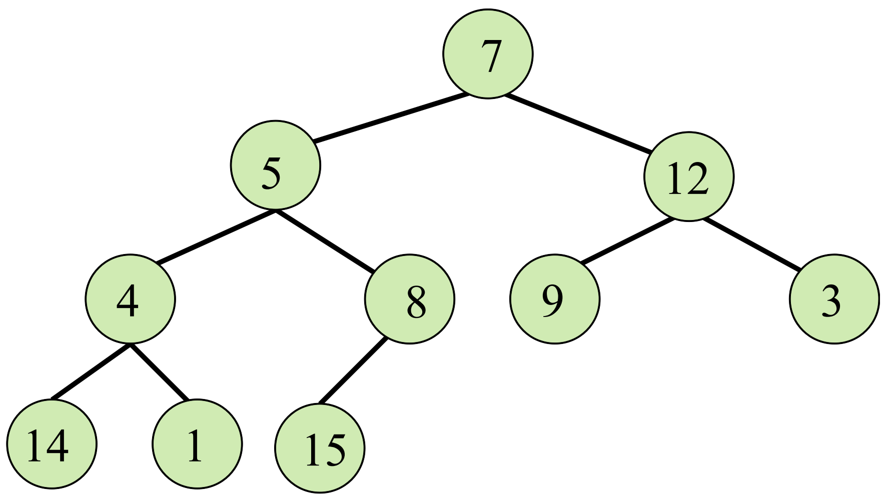
procedimento heapsort
    arranjar (n) $\longrightarrow O(n)$
    $m \leftarrow n$
    enquanto ($m > 1$) faça $\longrightarrow O(n)$
        troca($1,m$)
        $m \leftarrow m - 1$
        descer($1,m$) $\longrightarrow O(\log n)$

Complexidade: $O(n\log n)$

Heapsort

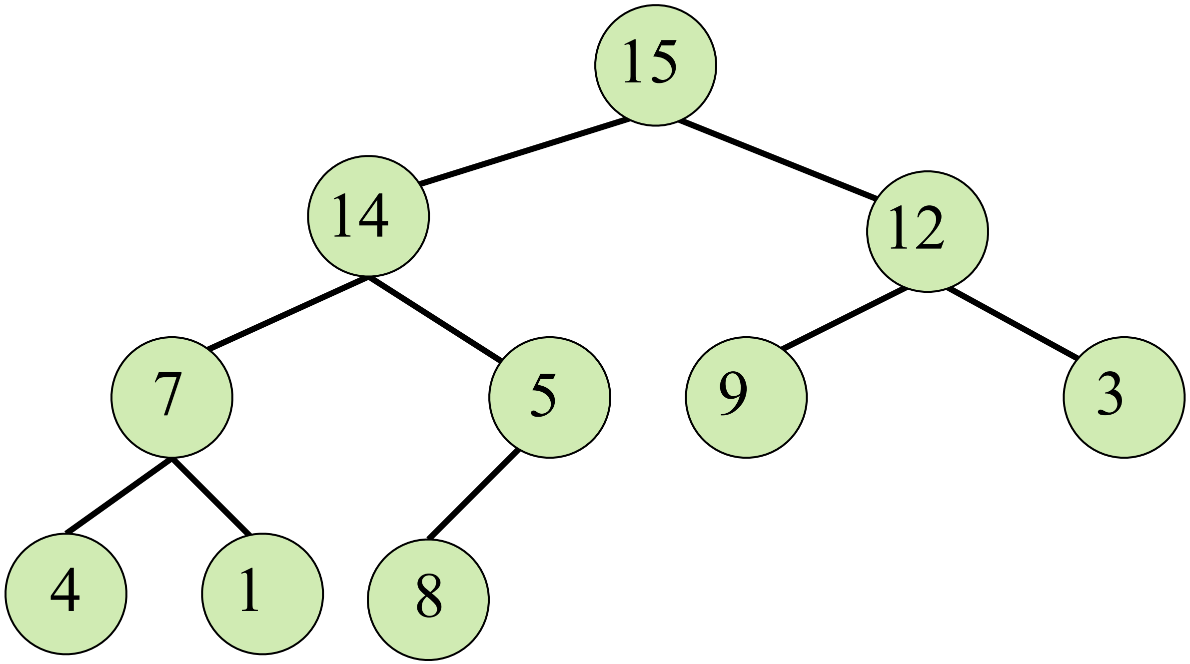
Exemplo: Aplicar Heapsort na seguinte lista.

7	5	12	4	8	9	3	14	1	15



Heapsort

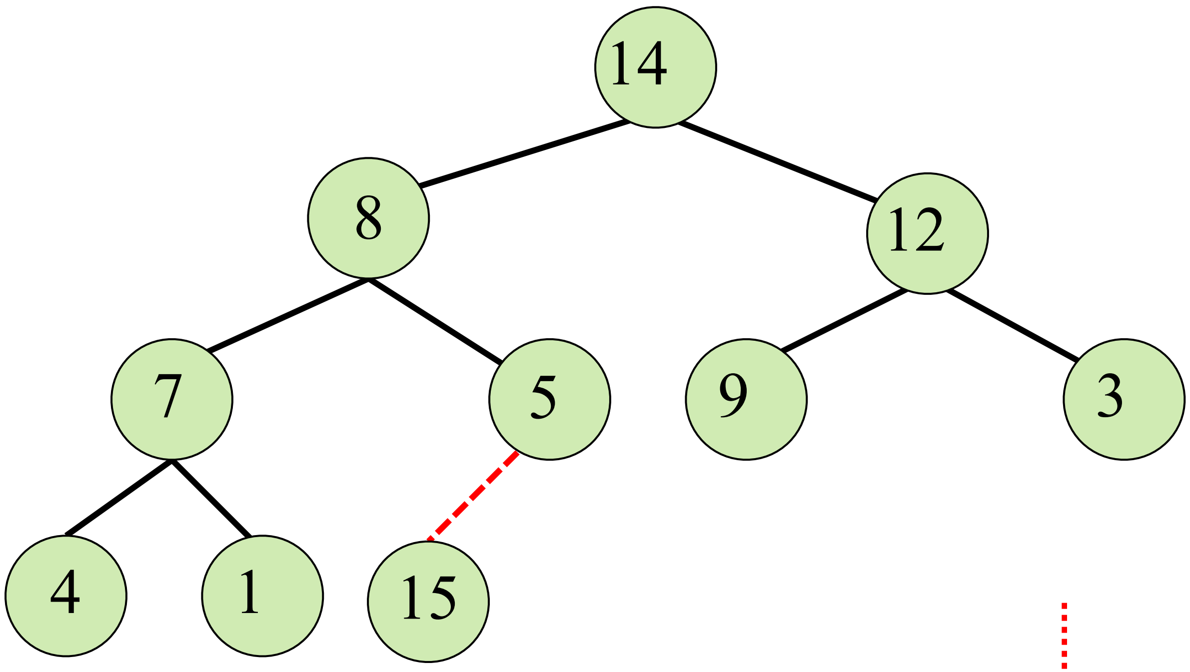
Arranjar(*n*)



15	**14**	**12**	**7**	**5**	**9**	**3**	**4**	**1**	**8**

Heapsort

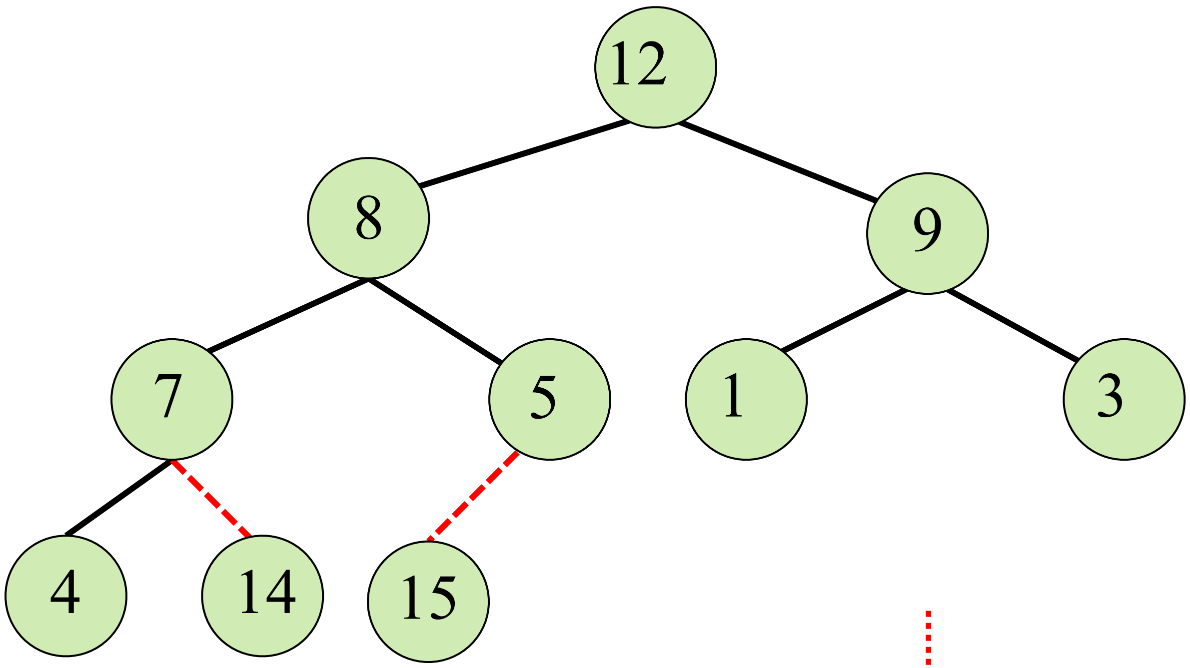
Trocar e descer



14	8	12	7	5	9	3	4	1	15

Heapsort

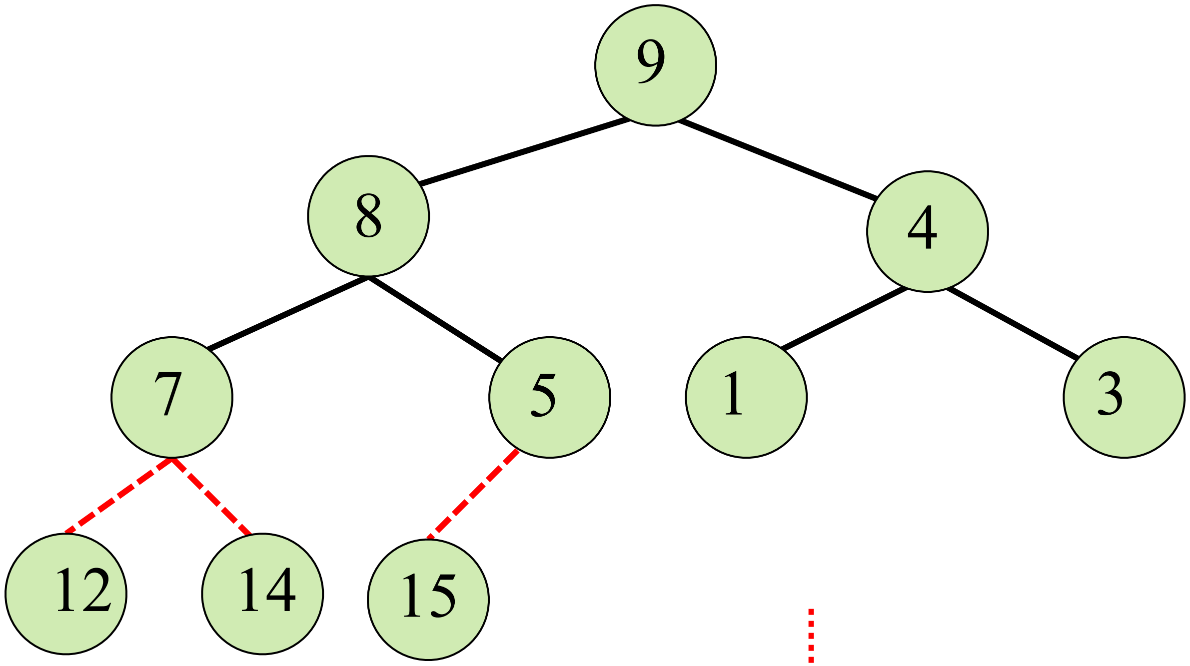
Trocar e descer



12	8	9	7	5	1	3	4	14	15

Heapsort

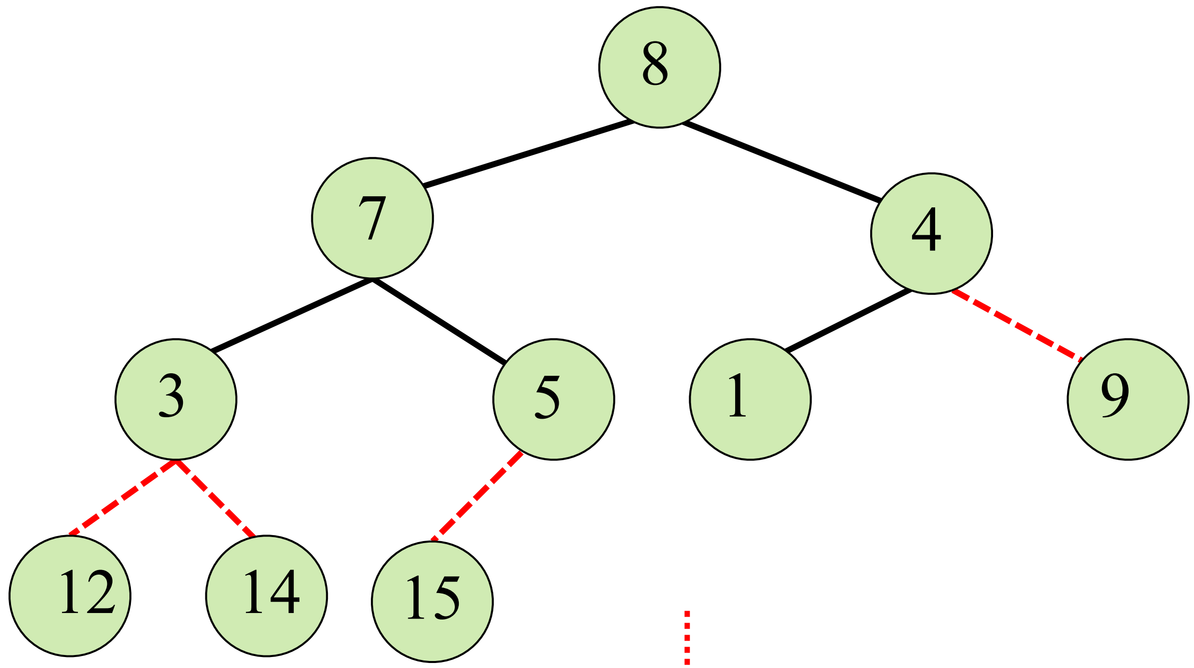
Trocar e descer



9	8	4	7	5	1	3	12	14	15

Heapsort

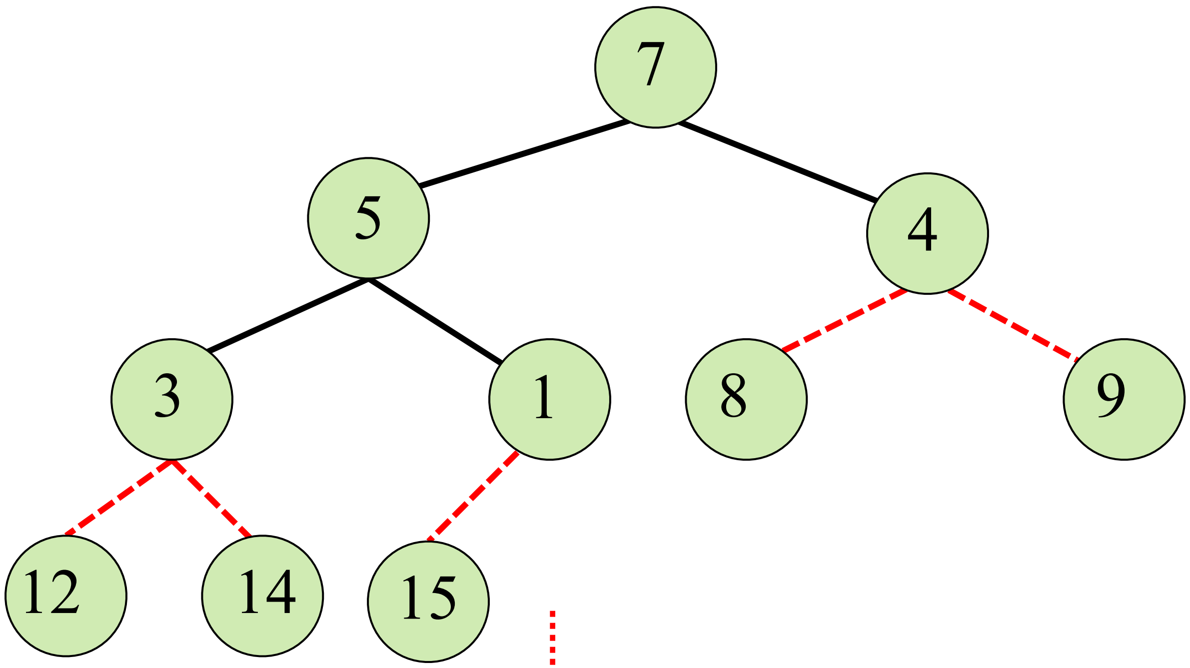
Trocar e descer



8	**7**	**4**	**3**	**5**	**1**	**9**	**12**	**14**	**15**

Heapsort

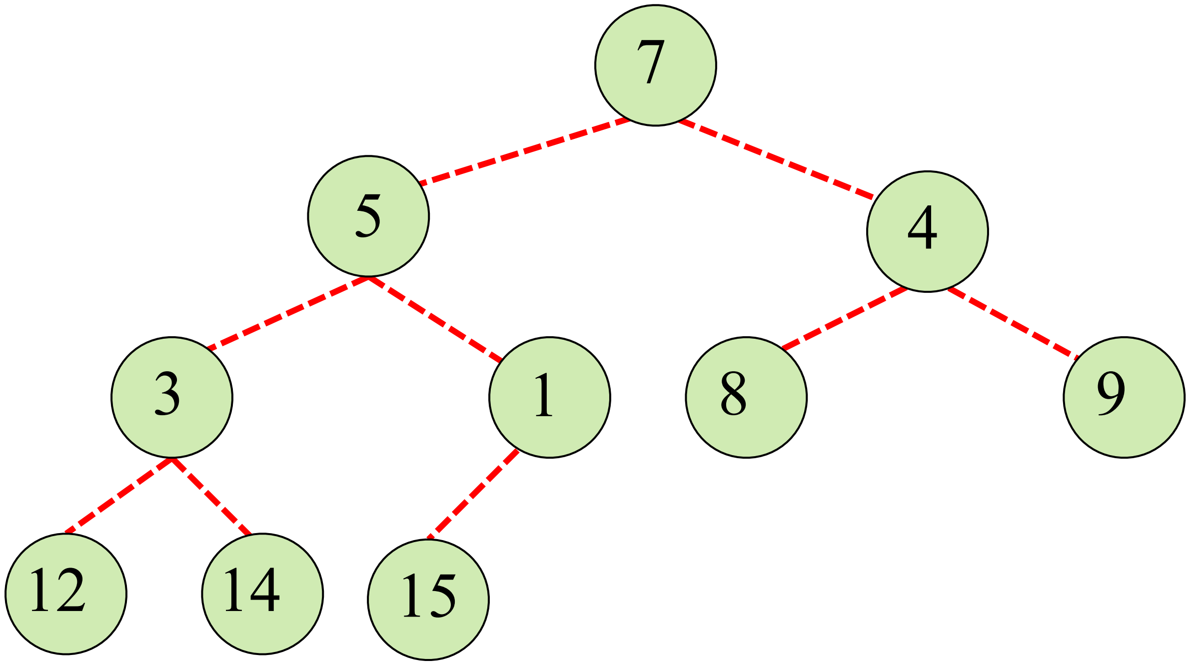
Trocar e descer



7	5	4	3	1	8	9	12	14	15

Heapsort

Final



1	3	4	5	7	8	9	12	14	15

Heapsort

Pior caso: **array já ordenado. O(n) para construir a heap max e O(nlong) para ordenar. Total: O(nlogn)**

Melhor caso: **array em ordem inversa (heap max trivialmente construída). O(nlong) para ordenar. Total: O(nlogn)**